7 A Challenge: The UGLY Language

We next propose a language design scenario in which a hypothetical language designer starts with the simple language in Figure 1 and extends it with various features in a rather adhoc manner. As the name of the resulting language, UGLY, may suggest, the purpose of this section is not to propose any novel interesting programming language (though one could write quite interesting and tricky UGLY programs), but rather to challenge the underlying language definitions framework. The ultimate goal of an ideal language definitional framework is, of course, to support such language design scenarios with minimal burden on the user. We argue that, for this scenario, K indeed requires minimal changes on existing definitions when adding the new features. We also discuss how other definitional frameworks fail to have this property. To make this language design scenario as reaslistic as possible, at each moment we pretend that the new feature is the last one to be added to the language. In other words, a feature to be added in the future cannot be used to justify a particular definitional choice at the current moment. For example, if one knew upfront that one wanted to eventually add references to one's language, then one may choose upfront to split the state into an environment and a store. However, we want to point out that if such a "radical" structural change requires one to revisit all or most of the existing definitions, then the underlying framework is far from ideal.

Variant 1 — increment. Let us add an increment construct, $\texttt{++}\_$ taking a variable, incrementing it in the store, and then returning the new, incremented value. In K one can do it easily as follows:

$$AExp ::= \ldots \mid \texttt{++}\, Var$$
$$k(\underline{\texttt{++}\,x}_{\sigma[x]+1}\rangle\ state(\underline{\quad\sigma\quad}_{\sigma[(\sigma[x]+1)/x]})$$

Since expression now have side effects, a big step definition of the previous variant of the language would need to be radically changed.

Variant 2 — unifying expressions. One annoying aspect of the current language is that one can only write/read integer values in the store, meaning also that one can only assign integer type expressions to variables. We would naturally like to eliminate this limitation and add arbitrary expressions to the language, and allow them to be assigned to variables. That means, in particular that we would like to unify the different categories of expressions, currently *AExp* and *BExp*, into only one syntactic category, say *Exp*; a type checker can be also easily defined in K to ensure that expressions are used properly, but we do not do it here. Also, we take the opportunity to add one more type of expressions, namely floats, which should, of course, enjoy the same first-class citizen rights of the other expressions. One more change to the language design is also in place here. In the original design a program was chosen to be a statement followed by an expression. Inspired by languages such as BC, suppose that we prefer at this stage to extend the expressions with a construct "*Exp* ::= ... | *Stmt*; *Exp*" and to eliminate programs all together, because we can replace them with expressions. There are quite some changes above; however, they can all be done very easily in K:

- Replace *AExp*, *BExp* and *Pgm* by *Exp* everywhere. This can be done either by editing the existing definitions mechanically (a tedious, but rather mechanic process), or better, by using a conventional *rename* module composition operator if the underlying implementation of K offers support for module composition. For example, with our Maude implementation of K, one can simply import the module "$\mathsf{Variant1} * (\mathsf{sort}\ AExp\ \mathsf{to}\ E,\ \mathsf{sort}\ BExp\ \mathsf{to}\ Exp,\ \mathsf{sort}\ Pgm\ \mathsf{to}\ E)$".

- Add float numbers to values, that is, add "*Val* ::= ... | *Float*", together with additional attributes for language constructs intended to also work with floats, namely $\_+\_$ and $\_\leq\_$:

  $\_+\_\ [strict\ extends\ \_+_{Int}\_\ extends\ \_+_{Float}\_]$
  $\_\leq\_\ [strict\ extends\ \_\leq_{Int}\_\ extends\ \_\leq_{Float}\_]$

A Simple Type Checker

$\mathit{TypeExp} := \mathit{int} \mid \mathit{float} \mid \mathit{bool}$, $\mathit{Type} ::= \mathit{TypeExp} \mid \mathit{stmt}$	$x.\mathsf{int} \to \mathit{int}$, $x.\mathsf{float} \to \mathit{float}$, $x.\mathsf{bool} \to \mathit{bool}$,
$\mathit{Exp}, \mathit{Stmt}, \mathit{Type} \leq K$	$\mathsf{++}\, x.\mathsf{int} \to \mathit{int}$, $\mathit{int} + \mathit{int} \to \mathit{int}$, $\mathit{float} + \mathit{float} \to \mathit{float}$,
$\mathit{Config} ::= \mathit{TypeExp} \mid [\![K]\!]_\tau$, $[\![\mathit{texp}]\!]_\tau = \mathit{texp}$	$\mathit{int} \leq \mathit{int} \to \mathit{bool}$, $\mathit{float} \leq \mathit{float} \to \mathit{bool}$,
$\_+\_$, $\_\leq\_$, $\_\mathsf{and}\_$, $\mathsf{not}\_$ $[\mathit{strict}]$	$\mathsf{not}\, \mathit{bool} \to \mathit{bool}$, $\mathit{bool}\, \mathsf{and}\, \mathit{bool} \to \mathit{bool}$,
$\_:=\_$, $\mathsf{if}\_\_\_$, $\mathsf{while}\_\_$, $\mathsf{halt}\_$ $[\mathit{strict}]$	$\tau := \tau \to \mathit{stmt}$, $\mathsf{if}\, \mathit{bool}\, \mathit{stmt}\, \mathit{stmt} \to \mathit{stmt}$,
$\_;\_$ $[\mathit{strict}(1)\ \_]$	$\mathsf{while}\, \mathit{bool}\, \mathit{stmt} \to \mathit{stmt}$, $\mathsf{halt}\, \mathit{texp} \to \mathit{stmt}$

Variation 3 – output

— add output — changes: modified CONFIGURATION to include an output configuration item — added module OUTPUT; modified LANGUAGE to include OUTPUT and eval to take stmts

$\mathit{ConfigItem} ::= k(K) \mid \mathit{state}(\mathit{State}) \mid \mathit{output}(\mathit{List}[\mathit{Val}])$
$\mathit{Config} ::= \mathit{List}[\mathit{Val}] \mid [\![\mathit{Stmt}]\!] \mid [\![\mathit{Set}[\mathit{ConfigItem}]]\!]$
$[\![s]\!] = [\![k(s)\ \mathit{state}(\cdot)\ \mathit{output}(\cdot)]\!]$
$[\![k(\cdot)\ \mathit{state}(\_)\ \mathit{output}(\mathit{vl})]\!] = \mathit{vl}$
$\mathsf{output}\_$ $[\mathit{strict}]$
$k(\underset{\cdot}{\underline{\mathsf{output}\, v}}\rangle\ \mathit{output}\langle\underset{v}{\underline{\cdot}})$

Variation 4(a) – λ with substitution

— add lambda expressions using substitution — assume substitution Exp[X ¡- Val] given, which avoids variable capture — one should be careful with side effects (this will be fixed in 06-lambda): — one should not assign or increment bound variables! — changes: added module LAMBDA-SUBST and imported it in LANGUAGE

$\mathit{Exp} ::= \ldots \mid \lambda\, \mathit{Var}.\mathit{Exp} \mid \mathit{Exp}\, \mathit{Exp}$ $[\mathit{strict}]$ (or $[\mathit{strict}(1)]$ for call-by-name)
$\mathit{Val} ::= \ldots \mid \lambda\, \mathit{Var}.\mathit{Exp}$
$k((\lambda x.e)\, v\rangle = k(e[v/x]\rangle$ (or $k((\lambda x.e)\, e'\rangle = k(e[e'/x]\rangle$ for call-by-name)

Variation 4(b) – λ with closures

— add lambda expressions, following a standard env/store state decomposition — changes: added module LOCATION; — modified CONFIGURATION to account for the state split and the nextLoc item — modified EXP, WRITE and LANGUAGE to account for the state split — added module LAMBDA and imported it

in LANGUAGE

$ConfigItem ::= k(K) \mid env(Env) \mid store(Store) \mid output(List[Val]) \mid nextLoc(Loc)$
$[\![e]\!] = [\![k(e)\ env(\cdot)\ store(\cdot)\ output(\cdot)\ nextLoc(0)]\!]$

$k(\underline{\ x\ \rangle}{\sigma[\rho[x]]}\ env(\rho)\ store(\sigma)$
$k(\underline{x := v\rangle}{\cdot}\ env(\rho)\ store(\underline{\ \sigma\ }{\sigma[v/\rho[x]]})$
$k(\underline{\ \texttt{++}\,x\ \rangle}{\sigma[\rho[x]]+1}\ env(\rho)\ store(\underline{\ \sigma\ }{\sigma[\sigma[\rho[x]]+1/\rho[x]]})$

$Exp ::= \ldots \mid \lambda\, Var.Exp \mid Exp\ Exp\ [strict]$
$Val ::= \ldots \mid closure(Var, Exp, Env)$
$K ::= \ldots \mid restore(Env)$
$k(\underline{\ \lambda x.e\ \rangle}{closure(x,e,\rho)}\ env(\rho)$
$k(\underline{closure(x,e,\rho)\ v\rangle}{e \curvearrowright restore(\rho')}\ env(\underline{\ \rho'\ }{\rho[l/x]})\ store(\underline{\ \sigma\ }{\sigma[v/l]})\ nextLoc(\underline{\ l\ }{s(l)})$
$k(v \curvearrowright \underline{restore(\rho)\rangle}{\cdot}\ env(\underline{\_}{\rho})$

Variant 5 – recursion

$Exp ::= \ldots \mid \mu\, Var.Exp$
$k(\mu x.e\rangle = k((\lambda x.e)\,(\mu x.e)\rangle$

Variant 6 – references

— add references: reference, dereference, assignment to locations — changes: removed module ASSIGNMENT; — added module REFERENCES and imported it in LANGUAGE

$Val ::= \ldots \mid Loc$
$Exp ::= \ldots \mid \mathsf{ref}\ Exp\ [strict] \mid *\ Exp\ [strict]$
$Stmt := \ldots \mid Exp := Exp\ [strict]$
$k(\underline{\mathsf{ref}\, v\rangle}{l}\ store(\underline{\ \sigma\ }{\sigma[v/l]})\ nextLoc(\underline{\ l\ }{s(l)})$
$k(\underline{*\,l\ \rangle}{\sigma[l]}\ store(\sigma)$
$k(\underline{l := v\rangle}{\cdot}\ store(\underline{\ \sigma\ }{\sigma[v/l]})$

Variant 7 – call with current continuation

— add callcc — changes: added module CALLCC and imported it in LANGUAGE

$Exp ::= \ldots \mid \mathsf{callcc}\ Exp\ [strict]$
$Val ::= \ldots \mid cc(K, Env)$
$k(\underline{\ \mathsf{callcc}\, v\ \curvearrowright k)}{v\ cc(k,\rho)}\ env(\rho)$
$k(\underline{cc(k,\rho)\, v \curvearrowright \_)}{v \curvearrowright k}\ env(\underline{\_}{\rho})$

Variant 8 – nondeterminism

— add nondeterminism via a random BExp — changed: added randBool to BEXP with two rules

$Exp ::= \ldots \mid \mathsf{randomBool}$
$\mathsf{randomBool} \to \mathsf{true}$
$\mathsf{randomBool} \to \mathsf{false}$

Variant 9 – aspects

$Stmt ::= \ldots \mid \mathsf{hole}$

$ConfigItem ::= \ldots \mid aspect(K) \mid stmt(K)$
$[\![s, c, s']\!] = [\![k(c)\ stmt(s)\ env(\cdot)\ store(\cdot)\ output(\cdot)\ aspect(s')\ nextLoc(0)]\!]$
$k(\underset{s}{\underline{\mathsf{hole}\rangle}}\ stmt(s)$
$k(\underset{closure(x,(s \curvearrowright e),\rho)}{\underline{\qquad \lambda x.e \qquad \rangle}}\ env(\rho)\ aspect(s)$

Variant 10 – concurrency with lock synchronization

$Stmt ::= \ldots \mid \mathsf{spawn}\ Stmt \mid \mathsf{acquire}\ Exp\ [strict] \mid \mathsf{release}\ Exp\ [strict]$
$ThreadItem ::= k(K) \mid env(Env) \mid holds(Set[Val \times Nat])$
$ConfigItem ::= \ldots\ (\text{remove } k, env) \mid thread(Set[ThreadItem]) \mid busy(Set[Val])$
$[\![s, c, s']\!] = [\![thread(k(c)\ env(\cdot))\ stmt(s)\ store(\cdot)\ output(\cdot)\ aspect(s')\ nextLoc(0)]\!]$
$thread\langle k(\underset{\cdot}{\underline{\mathsf{spawn}\ s\rangle}}\ env(\rho)\rangle\ \underset{thread(k(s)\ env(\rho)\ holds(\cdot))}{\underline{\qquad \cdot \qquad}}$
$\underset{\cdot}{\underline{thread\langle k(\cdot)\ holds(lc)\rangle}}\ busy(\underset{ls - lc}{\underline{\quad ls \quad}})$
$k(\underset{\cdot}{\underline{\mathsf{acquire}\ v\rangle}}\ holds\langle(v, \underset{s(n)}{\underline{\quad n \quad}})\rangle$
$k(\underset{\cdot}{\underline{\mathsf{acquire}\ v\rangle}}\ holds\langle\underset{(v,0)}{\underline{\quad \cdot \quad}}\rangle\ busy(\underset{ls\ v}{\underline{\quad ls \quad}})\ \texttt{if}\ v \notin ls$
$k(\underset{\cdot}{\underline{\mathsf{release}\ v\rangle}}\ holds\langle(v, \underset{n}{\underline{s(n)}})\rangle$
$k(\underset{\cdot}{\underline{\mathsf{release}\ v\rangle}}\ holds\langle\underset{\cdot}{\underline{(v,0)\rangle}}\ busy\langle\underset{\cdot}{\underline{v\rangle}}$

Variant 11 – rendez-vous synchronization

$Stmt ::= \ldots \mid \mathsf{rv}\ Exp\ [strict]$
$k(\underset{\cdot}{\underline{\mathsf{rv}\ v\rangle}}\ k(\underset{\cdot}{\underline{\mathsf{rv}\ v\rangle}}$

Variant 12 – self-generation of code

$Exp ::= \ldots \mid \mathsf{quote}\ Exp \mid \mathsf{unquote}\ Exp \mid \mathsf{eval}\ Exp\ [strict]$
$Val ::= \ldots \mid code(K)$
$K ::= \ldots \mid \mathsf{quote}\ List[K] \mid code(List[K]) \mid K \frown K\ [strict] \mid \mathit{Id}(List[K])\ [strict(2)] \mid K \blacksquare K\ [strict]$

$k(\mathsf{quote}(k)\rangle = k(quote(0, k)\rangle$
$quote(n, k_1 \curvearrowright k_2) = quote(n, k_1) \frown quote(n, k_2)$ $\quad code(k_1) \frown code(k_2) = code(k_1 \curvearrowright k_2)$
$quote(n, f(kl)) = \mathit{f}(quote(n, kl))$ if $f \neq \mathsf{quote}, \mathsf{unquote}$ $\quad \mathit{f}(code(kl)) = code(f(kl))$
$quote(n, \mathsf{quote}(k)) = \mathsf{quote}(quote(s(n), k))$
$quote(0, \mathsf{unquote}(k)) = k$
$quote(s(n), \mathsf{unquote}(k)) = \mathsf{unquote}(quote(n, k))$

$quote(n, (k, kl)) = quote(n, k) \blacksquare quote(n, kl)$ if $kl \neq \cdot$ $\quad code(k) \blacksquare code(kl) = code(k, kl)$
$quote(n, k) = code(k)$ if $k \in Val \cup Var$ $\quad \mathsf{eval}\ code(k) = k$